Programación Python para Machine Learning

Lección 7: Support Vector Machines y Naive Bayes.

eip ESCUELA INTERNACIONAL POSGRADOS

ÍNDICE

Lección 7: Support Vector Machines y Naive Bayes.

1. Introducción

En la siguiente lección, se van a exponer dos tipos de modelos que se pueden utilizar en proyectos de Machine Learning donde haya que enfrentarse a un problema de clasificación supervisada. Concretamente, se estudiarán los principios, aplicación y consideraciones prácticas de dos algoritmos de carácter no lineal de clasificación que pueden aplicarse en un conjunto de datos:

- Las Support Vector Machines o SVM.
- Los clasificadores Naive Bayes.

Objetivos

- Conocer los principios de las Máquinas de Vectores Soporte (SVM).
- Saber implementar en Python modelos de SVM para resolver problemas de clasificación y regresión.
- Aprender las bases de los modelos Naive Bayes.
- Explorar las herramientas para implementar en Python modelos Naive Bayes para resolver problemas de clasificación.

2. Support Vector Machines (SVM)

En esta sección se estudiará brevemente qué son las Support Vector Machines (SVM, Máquinas de Vectores Soporte) y cómo se puede implementar en Python.

Los SVM son un tipo de modelo supervisado de Machine Learning para resolver de clasificación y su carácter es no lineal. Los SVM, propuestas en la década de los 60 y perfeccionadas en los 90, son en la actualidad tremendamente populares debido a su más que notable rendimiento.

2.1. Principios del modelo SVM

En el caso de datos linealmente separables, otros algoritmos de Machine Learning tratan de resolver el problema buscando un límite que divida las instancias de modo que se minimice el error de clasificación. Esto conduce a que pueda haber varios límites que dividan las instancias como puede observarse en la Ilustración 1.

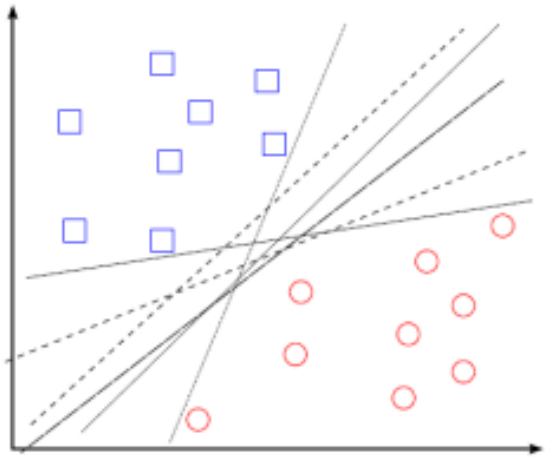

Ilustración 1: Múltiples resoluciones lineales

Sin embargo, los SVM tratan de buscar un límite que maximice la distancia de los patrones más cercanos de las dos clases. Es decir, no solo encuentra un límite de separación, si no que se decide por el límite de decisión óptimo (véase Ilustración 2).

Los patrones que se encuentran en el límite entre las dos clases y que, por tanto, se utilizan para encontrar la máxima separación, se denominan vectores soporte.

Normalmente, dicha separación, se realiza mediante una transformación de las dimensiones de entrada en un espacio de mayor dimensionalidad, espacio de Hilbert, en la que las clases son más separables, utilizando lo que se conoce matemáticamente como el truco kernel o *kernel trick*. Se trata por tanto de un modelo que permite una separación no lineal de los patrones.

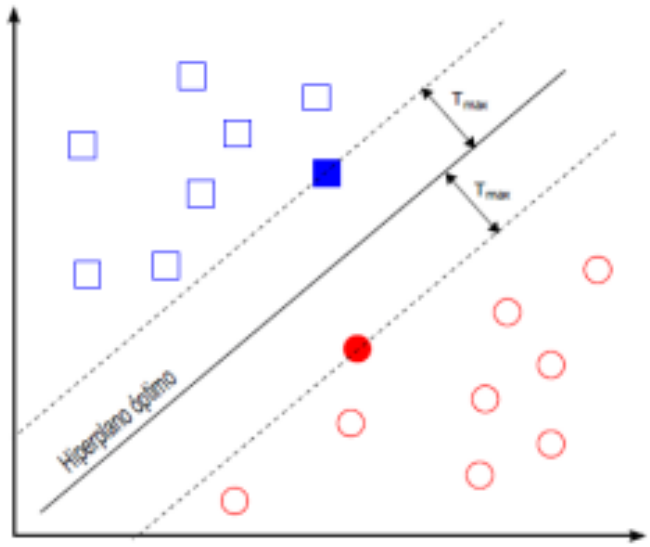

Ilustración 2: Resolución lineal óptima.

2.2. Implementación en Python de un modelo SVM

Se puede llevar a cabo el proceso de construcción de un modelo de clasificación SVM en Python utilizando la clase `SVC` del módulo `svm` de scikit-learn. Se hará uso del conjunto de datos *Indian-Liver-Patient.*

En este caso, hay que recordar que cuando se trata con un modelo de SVM, todas las variables deben ser de naturaleza numéricas.

Por tanto, toda variable categórica en el conjunto de datos debe ser convertidas en numéricas del modo más adecuado posible.

```
from pandas import read_csv
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import confusion_matrix, balanced_accuracy_score
import random, time, pandas, numpy

seed=random.seed(time.time())
filename = '../datasets/Indian-Liver-Patient.csv'

col_names = ['age', 'gen', 'tbili', 'dbili', 'alkphos', 'sgpt',
'sgot', 'tp', 'alb', 'ag' ,'class']
data = read_csv(filename, names=col_names)

X = data[data.columns[:-1]].fillna(0)
Y = data['class']

X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size =
0.3, random_state=seed)

num_X_train=X_train.select_dtypes(include=['int', 'float'])
num_X_test=X_test.select_dtypes(include=['int', 'float'])

stdScaler = StandardScaler()
stdScaler.fit(num_X_train)
resc_X_train  =
pandas.DataFrame(stdScaler.transform(num_X_train),columns=num_X_train.
columns )
resc_X_test =
pandas.DataFrame(stdScaler.transform(num_X_test),columns=num_X_test.co
lumns )

cat_X_train=X_train.select_dtypes(include=['object'])
cat_X_test=X_test.select_dtypes(include=['object'])

ohe =OneHotEncoder()
ohe.fit(cat_X_train)
resc_X_train[ohe.get_feature_names()] =
ohe.transform(cat_X_train).toarray()
resc_X_test[ohe.get_feature_names()]=
ohe.transform(cat_X_test).toarray()


modelo = SVC(C=1, class_weight='balanced', kernel='rbf', gamma=0.1)
modelo.fit(resc_X_train, y_train)
y_pred = modelo.predict(resc_X_test)

cm =  confusion_matrix(y_test, y_pred)
acc=  balanced_accuracy_score(y_test, y_pred)
print(cm, acc)
```

2.3. Consejos prácticos sobre las SVM

Ajuste de parámetros

El rendimiento de un modelo SVM está tremendamente condicionado a la configuración de sus parámetros. En particular, hay tres aspectos que deben ser tenidos muy en cuenta:

1. El kernel utilizado: es la función de kernel a utilizar. En scikit-learn, los kernel disponibles son:
    a. Kernel Lineal: `kernel= 'linear'`
    b. Kernel Polinomial: `kernel= 'poly'`
    c. Kernel Gausiano (por defecto): `kernel= 'rbf'`
    d. Kernel Sigmoide: `kernel= 'sigmoid'`
    e. Kernel precomputado: `kernel= 'precomputed`
2. El parámetro C: El parámetro de regularización. Tiene que ser estrictamente positivo. Por defecto `C=1`.
3. El parámetro gamma: es el coeficiente del kernel. Solo utilizado en los kernel polinomial, gaussiano y sigmoide.

Para obtener una configuración ideal del modelo, la práctica más extendida es hacer una búsqueda en *grid,* en la que se comprueben diversas combinaciones de estos parámetros.

El rendimiento de los modelos SVM, como la mayoría de clasificadores, sufre cuando son entrenados con conjuntos de datos desequilibrados. Por ello, se debe considerar, bien equilibrar el conjunto de datos de entrenamiento, o bien utilizar el ajuste del parámetro `class_weight='balanced'`.

Clasificación multiclase

En su naturaleza, SVM resuelve problemas de clasificación binarios. Sin embargo, existen estrategias para poder escalar sus principios a problemas en los que los patrones pertenecen a más de dos clases.

- La estrategia Uno-contra-Todos (OVR): divide una clasificación multiclase en un problema de clasificación binaria de los patrones de cada clase contra los patrones del resto de clases en conjunto. Si hay *n* clases, se generan *n* modelos. La predicción del patrón será en base a la máxima salida entre todos los modelos.
- La estrategia Uno-contra-Uno (OVO): divide una clasificación multiclase en un problema de clasificación binaria por cada par de clases, generándose n(n-1)/2 modelos. La predicción del patrón será en base un proceso de votación de todos los modelos.

Por defecto `SVC` de scikit-learn trae por defecto configurada la estrategia OVR, siendo totalmente transparente al programador esta característica. En caso de que se necesite la utilización de la estrategia OVO, se puede configurar mediante el parámetro `decision_function_shape= 'ovo'`.

2.4. Problemas de regresión utilizando SVM

Las SVM se desarrollaron para resolver problemas de clasificación binarios. Sin embargo, la técnica fue extendida para los problemas de predicción de valores reales, es decir, para resolver problemas de regresión. Es posible generar un modelo de regresión SVM en Python utilizando la clase `SVR` del módulo `svm` de scikit-learn. Se hará uso del conjunto de datos *housing.*

Como en el resto de casos, hay que recordar que cuando se trata con un modelo de SVM para regresión, todas las variables deben ser de naturaleza numéricas. Por tanto, toda variable categórica en el conjunto de datos debe ser convertidas en numéricas del modo más adecuado posible.

```
from pandas import read_csv
import numpy, random, time
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.svm import SVR
from sklearn.metrics import mean_squared_error

seed=random.seed(time.time())
filename = '../datasets/housing.csv'

col_names = ['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS',
'RAD', 'TAX', 'PTRATIO', 'B', 'LSTAT', 'MEDV']
data = read_csv(filename, names=col_names, sep=" ")

X = data[data.columns[:-1]]
Y = data['MEDV']

X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size =
0.3, random_state=seed)

stdScaler = StandardScaler().fit(X_train)
X_train  = stdScaler.transform(X_train)
X_test   = stdScaler.transform(X_test)

modelo = SVR()
modelo.fit(X_train, y_train)
y_pred = modelo.predict(X_test)

rmse = (numpy.sqrt(mean_squared_error(y_test, y_pred)))
print(rmse)
```

3. Naive Bayes

Uno de los más importantes teoremas de la probabilidad dentro de la estadística es el Teorema de Bayes. Considerado como la base del razonamiento deductivo, este teorema se centra en determinar la probabilidad de que ocurra un evento teniendo en cuenta el conocimiento previo de las condiciones que podrían estar relacionadas con el evento.

El clasificador Naive Bayes utiliza los principios de este teorema para construir un modelo de clasificación en Machine Learning simple y con gran rendimiento. Esta sección presenta una descripción general sobre cómo funciona Naive Bayes y cómo utilizar su implementación incluida en la librería scikit-Learn de Python.

3.1. Principios del modelo Naive Bayes

El teorema de Bayes define:

$$P(A/B) = \frac{P(B/A) \cdot P(A)}{P(B)}$$

Siendo

- $P(A/B)$ la probabilidad de A dándose el evento B. Probabilidad a posteriori.
- $P(B/A)$ la probabilidad de que se produzca B dado que la hipótesis A es verdadera.
- $P(A)$ la probabilidad a priori.
- $P(B)$ la probabilidad marginal del evento B.

Po tanto, el Teorema de Bayes permite realizar una deducción de que ocurra un evento en base al conocimiento previo de las observaciones que puedan implicarlo. Para aplicar este teorema a cualquier problema, se necesitan calcular los dos tipos de probabilidades que aparecen en la fórmula, las probabilidades marginales y las condicionadas.

En el teorema, P(A) representa las probabilidades de cada evento. En el clasificador Naive Bayes, se pueden interpretar estas probabilidades simplemente como la frecuencia de cada instancia del evento dividida por el número total de instancias, la probabilidad de cada clase.

La segunda probabilidad es P(A|B), que representa las probabilidades condicionales de un evento A dado otro evento B. En el clasificador Naive Bayes, estas codifican la probabilidad posterior de A ocurriendo cuando B es verdad.

De forma general, se puede hacer una ampliación del Teorema de Bayes del siguiente modo:

$$P\left(A/_{B_1, B_2 \dots B_n}\right) = \frac{P\left(B_1, B_2 \dots B_n/_A\right) \cdot P(A)}{P(B_1, B_2 \dots B_n)}$$

Completar los cálculos de la expresión para todas las clases posibles es muy costoso computacionalmente. Por tanto, se necesitan hacer una serie de suposiciones que simplifiquen los cálculos. Los clasificadores Naive Bayes asumen que todas las características son independientes entre sí, por eso lo de ingenuo. Además, dependiendo de la asunción que se haga sobre la distribución que siguen las variables de entrada, se considerará un tipo u otro de clasificador Naive Bayes: gaussiano, multinomial, de Bernoulli...

3.2. Implementación en Python de un modelo Naive Bayes

Se puede llevar a cabo el proceso de construcción de un modelo de clasificación Naive Bayes en Python utilizando las del módulo `naive_bayes` de scikit-learn. Se hará uso del conjunto de datos *chess.*

En este caso, hay que recordar que, aunque los algunos modelos Naive Bayes admiten trabajar con variables categóricas, en scikit-learn todas las variables deben ser de naturaleza numéricas.

Por tanto, toda variable categórica en el conjunto de datos debe ser convertidas en numéricas del modo más adecuado posible.

```
from pandas import read_csv
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, balanced_accuracy_score
import random, time
from sklearn.naive_bayes import CategoricalNB, GaussianNB
from sklearn.preprocessing import LabelEncoder

seed=random.seed(time.time())

filename = '../datasets/kr-vs-kp.data'
col_names = list(map(str,range(36))) + ['class']
data = read_csv(filename, names=col_names)

le = LabelEncoder()
data = data.apply(le.fit_transform)

input_data = data[data.columns[:-1]]
X= input_data
Y = data['class']

X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size =
0.3, random_state=seed)

modelo = GaussianNB()
modelo.fit(X_train, y_train)
y_pred = modelo.predict(X_test)

cm =  confusion_matrix(y_test, y_pred)
acc=  balanced_accuracy_score(y_test, y_pred)
print(cm, acc)

modelo =CategoricalNB()
modelo.fit(X_train, y_train)
y_pred = modelo.predict(X_test)

cm =  confusion_matrix(y_test, y_pred)
acc=  balanced_accuracy_score(y_test, y_pred)
print(cm, acc)
```

3.3. Consejos prácticos sobre los modelos Naive Bayes

Naive Bayes es un algoritmo que, al ser simple y fácil de calcular, podría ser útil frente a modelos más complejos cuando la cantidad de datos es limitada. Además, puede trabajar con datos tanto numéricos como categóricos, incluso en algún caso, el modelo Naive Bayes gaussiano puede utilizarse para realizar regresiones.

Por el contrario, debido a los principios del teorema, presenta problemas cuando falta alguna combinación “etiqueta de clase-determinado valor de atributo” en los datos de entrenamiento, puesto que la estimación de probabilidad basada en la frecuencia será cero. Esto unido a la suposición de independencia, cuando se multiplican todas las probabilidades se obtendrá cero.

Hay numerosos escenarios donde se ejemplifica el éxito de la aplicación de clasificadores Naive Bayes para resolución de problemas. El caso considerado más clásico es la clasificación de documentos o procesamiento de textos. El modelo funciona bien en problemas que involucran palabras clave como características, pero no tanto cuando existe una relación fuerte entre los términos o palabras.

4. Puntos clave

En esta lección hemos aprendido:

- Acercarse a los principios de las Máquinas de Vectores Soporte (SVM).
- Conocer el proceso de implementación en Python modelos de SVM para resolver problemas de clasificación y regresión.
- Entender las bases conceptuales de los modelos Naive Bayes.
- Utilizar las herramientas para implementar en Python modelos Naive Bayes para resolver problemas de clasificación.